Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23.М07-мм

Разработка высоконагруженной системы формирования бизнес-отчётности формата XBRL

Киреев Андрей Андреевич

Отчёт по производственной практике
в форме «Производственное задание»

Научный руководитель:

доцент кафедры Системного Программирования Луцив Д.В.

Консультант:

руководитель отдела «Смарт-прайсинг» Гришин В.С.

Место работы: ООО «ОЗОН Технологии»

Санкт-Петербург

2025

Оглавление

Введение

Организациям и банкам необходимо каждый квартал отчитываться в Центральный банк Российской Федерации (он же Банк России, Центробанк, ЦБ РФ), сообщая историю финансовых взаимодействий, сведения по кредитным источникам и контрагентам. Оформление этих сведений сводится к формированию общего для всех XBRL-отчёта. XBRL — это расширяемый язык деловой отчётности (от англ. extensible Business Reporting Language), формат передачи регуляторной, финансовой и иной отчётности [1]. XBRL-отчёт — это файл, передаваемый в Центральный банк Российской Федерации (он же Банк России, Центробанк, ЦБ РФ) отчитывающейся финансовой организацией, в котором указывается деятельность этой организации в формате установленным правилами и требованиями ЦБ РФ [2]. Т.к. данные для передачи, процесс их хранения и взаимодействия индивидуальны - нет готового универсального решения для каждой отдельной компании. Это является основной предпосылкой к созданию автоматизированной системы сбора XBRL-отчёта.

В работе прошлого семестра были уточнены конкретные отчёты, необходимые для формирования в системе:

1. Форма 0420754 «Сведения об источниках формирования кредитных историй» [3].

- Раздел II. Сведения о записях кредитных историй и (или) иных данных, передаваемых источниками формирования кредитных историй.
- Раздел III. Сведения об источниках формирования кредитных историй, которым были уступлены права требования, не предоставляющих сведения в бюро кредитных историй.
- Раздел IV. Сведения о передаче источниками формирования кредитных историй недостоверных сведений в бюро кредитных историй.

2. Форма 0420755 «Сведения о пользователях кредитных историй» [4].

- Раздел II «Сведения о количестве запросов, полученных бюро кредитных историй».
- Раздел III. Сведения об отказах бюро кредитных историй в исполнении запросов пользователей кредитных историй, лиц, запросивших кредитный отчёт.
- Раздел V. Сведения о запросах пользователей кредитных историй на получение кредитных отчётов субъектов кредитных историй.

3. Форма 0420762 «Реестр контрагентов» [5].

Из-за того, что создание отчётов на момент реализации данной системы проводился с помощью обученных сотрудников — имелась необходимость в системе, автоматизирующей весь процесс. Система работает самостоятельно в фоне и имеет возможность взаимодействия со специально обученными пользователями, которые могут манипулировать автоматизированной в нестандартных или экстренных случаях.

В процессе первичной эксплуатации системы появилась необходимость в стабильной работе системы под высокой нагрузкой. Несмотря на то, что приложение преимущественно сосредоточено на автоматизированной работе — процесс работы не должен деградировать при наличии нагрузки. Также для быстрого и удобного процесса проверки входных и выходных данных и доступа к предопределённым способам управления сотрудникам понадобился пользовательский интерфейс к созданной системе.

1. Постановка задачи

Целью работы является добавление пользовательского интерфейса и возможностей эксплуатации под высокой нагрузкой. Для достижения цели были поставлены следующие задачи:

1. Продумать логику работы микросервисов с учётом обеспечения высокой нагрузки. Провести и проанализировать нагрузочные тесты для каждого микросервиса.
2. Выбрать технологию для реализации пользовательского интерфейса.
3. Реализовать пользовательский интерфейс.

2. Обзор ранее реализованного функционала

В данной работе продолжается процесс создания системы XBRL-отчётности, которая реализовывалась в работах предыдущих семестров. Исходя из этого стоит привести краткие сведения по основной информации о приложении и о проделанной ранее работе.

Система представляет из себя Backend-проект, разработанный на языке Golang[6]. Приложение должно автоматически по заданному расписанию собирать комплект XBRL-отчётов. Данные для сборки берутся из ряда источников: хранилище данных Ceph S3[7], базы данных PostgreSQL[8] и Oracle[9]. Данные также автоматически загружаются и сохраняются в нужном формате в собственные базы системы, не допуская потерю целостности.

Система разделена на два микросервиса ввиду разности источников. Первый микросервис занимается автоматической загрузкой и парсингом в базу данных квитанций, хранящихся в Ceph S3 как файлы формата JSON. Второй микросервис занимается как автоматической загрузкой необходимых данных из баз источников в целевую базу проекта, так и формированием XBRL-отчётов. Для реализации автоматизации был написан планировщик, работающий на основании расписания в виде CRON[1]-строк, встроенная в память очередь задач и утилита определения приоритетности задач, поступающих одновременно на исполнение. А обеспечение поддержки работы приложения в нескольких экземплярах реализовано с помощью написанного динамического выбора лидера, контролирующего процесс загрузки. Также в предыдущих работах рассчитаны показатели вводимых баз данных и внесено некоторое количество технического функционала, а именно: реализовано взаимодействие с системой по API, добавлены метрики Prometheus, написаны юнит-тесты до 50% порога, создан Dockerfile для упрощения деплоя микросервисов.

[1] CRON - программа, предназначенная для выполнения заданий в определенное время, или через определенные промежутки времени.

3. Поддержка высокой нагрузки

3.1. Основные определения

Для корректного анализа способностей системы стоит для начала определиться с основными формулировками. Высоконагруженные приложения — это программные системы, способные эффективно обрабатывать и обслуживать большое количество одновременных запросов и пользователей, обеспечивая при этом стабильную работу[10]. Такие приложения часто используются в сферах, где критически важны высокая производительность и надежность, например, в онлайн-банкинге, социальных сетях, стриминговых сервисах и крупных интернет-магазинах.

Одними из показателей высокой нагрузки, по которым анализируют возможности приложения, являются следующие характеристики[11]:
1. RPS (request per second) – количество запросов в секунду, которое способно выдержать приложение.
2. «Response Time» – время отклика приложения на входящие запросы.
3. «Failed requests» – количество ошибочных ответов от системы. При разработке необходимо минимизировать этот показатель.

Основной метрикой, которой оперируют разработчики высоконагруженных сервисов – это RPS. Определение количества запросов в секунду, которое должно выдерживать приложение, зависит от множества факторов, включая внутреннюю специфику приложения, архитектуру, используемое оборудование и запланированные требования к нагрузке. Абсолютной градации RPS для определения какое приложение является низконагруженным, а какое высоконагруженным – не существует. Сервисы электронной коммерции или ведущих маркетплейсов могут стабильно выдерживать от нескольких десятков тысяч RPS до нескольких миллионов[12]. В то же время для сайтов небольших интернет-магазинов или образовательных порталов длительная стабильная нагрузка в 100 RPS может оказаться критичной.

Поэтому для того, чтобы определить порог RPS для конкретной системы – необходимо знать количество одновременно взаимодействующих с системой пользователей.

Оценить стабильность сервиса под разным количеством RPS можно с помощью проведения нагрузочного тестирования. Нагрузочное тестирование — это тип тестирования производительности, при котором проверяется, как система ведёт себя при заданной нагрузке, то есть имитирует ситуацию, когда множество пользователей одновременно выполняют некоторые действия, а именно отправляют HTTP-запросы[13]. Нагрузочное тестирование отображает соотношение показателей RPS, «Response Time» и «Failed requests». Также нагрузочное тестирование может выявить ошибки и неточности в разработанной системе.

Для выбора удобного инструмента проведения нагрузочного тестирования стоит рассмотреть сравнительную таблицу:

Инструмент	**Язык / UI**	**Преимущества**	**Недостатки**
k6[14]	JavaScript (CLI)	1. Высокая производительность. 2. Сценарии пишутся на JavaScript. 3. Свободная интеграция с Grafana.	1. CLI-ориентирован, не подходит для GUI-тестеров. Может быть сложен для начинающих. 2. Популярен при тестировании высоконагруженных сервисов.
Gatling[15]	Scala (CLI)	1. Высокая производительность. 2. Поддержка сложных сценариев.	1. Требует знания языка Scala. 2. Непопулярный в сообществе тестеров.
Locust[16]	Python (CLI/Web UI)	1. Прост в использовании. 2. Веб-интерфейс.	1. Малая производительность по сравнению с k6.
wrk2[17]	C (CLI)	1. Высокая производительность. 2. Малая нагрузка на CPU.	1. Ограниченный функционал. 2. Нет отчетов/визуализации.

Таблица 1 – Инструменты нагрузочного тестирования

Наиболее предпочтительной утилитой для проведения нагрузочного теста является «k6», поскольку «Gatling» требует знание специфичного языка Scala, «Locust» может не показывать необходимых результатов при высокой нагрузки, а «wrk2» не визуализирует итоговый результат. Соответственно первый инструмент и будет выбран для работы. Для запуска k6 нужно описать сценарии нагрузки на языке JavaScript и запустить их командой:

```
K6_WEB_DASHBOARD=true k6 run example.js
```

Также стоит определить некоторые характеристики тестирования:

1. Каждый микросервис развёрнут на одном поде Kubernetes и работает на 1 ядре процессора. Это сделано для выявления поведения приложения в минимальных условиях.
2. Процессор машины, на которой развёрнут Kubernetes: Intel Core i3 12100 (доступно 4 ядра).
3. CPU Usage ограничен 90% для предотвращения падения машины.
4. Параметры используемых баз данных и Ceph S3 близки к «боевым» и поддерживают десятки тысяч RPS.

3.2. Проведение нагрузочного тестирования «первого» микросервиса

Как упоминалось в работах предыдущих семестров – «первый» микросервис (здесь и далее ПМ) отвечает за автоматический перенос данных из хранилища Ceph S3 в собственную базу данных PostgreSQL. Это является ключевой логикой микросервиса. Однако, у ПМ также предусмотрены и POST-запросы для загрузки конкретных файлов в базу, например, для анализа ошибки загрузки файла или загрузки в базу пропущенного файла. Несмотря на то, что при стандартной эксплуатации сервиса эти запросы будут использоваться единично и крайне редко – стоит определить пиковые возможности ПМ и устранить ошибки, если такие будут обнаружены.

Запросы, которые будут протестированы («fillDbByFirstNotice», «fillDbBySecondNotice», «fillDbByAlarm») – не имеют сложной логики: каждый

из них получает в теле запроса параметр «jsonFileName» — наименование JSON-файла, скачивает этот файл из Ceph S3, парсит согласно внутренней логике и вставляет записи в нужные таблицы в БД. Поэтому сложных оптимизаций здесь проводить не рационально.

Максимальное количество потенциальных пользователей, которые в момент разбора внештатной ситуации могут начать пользоваться ПМ, ограничивается числом менее 20 человек. Это примерная численность сотрудников отдела разработки и отдела бизнес-заказчиков. Будем использовать худший сценарий, когда каждый пользователь делает запрос каждую секунду на каждую ручку, что в реальности практически невозможно. Соответственно, преодоление порога в 60 RPS будет свидетельствовать о высоконагруженности ПМ.

Далее разберём код нагрузочного сценария, листинг которого представлен в приложении 1. В нём есть основной сценарий плавного поэтапного увеличения RPS до заданного значения:

```
firstNoticesRequests: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 100 },
                { duration: '18m', target: 100 },
                { duration: '20s', target: 0 },
            ],
            exec: 'firstNoticesRequests',
        }
```

Также в коде есть 3 функции, каждая из которых описывает логику запроса к ПМ и проверку статуса его ответа. Каждая из них обращается к банку данных наименований квитанций, берёт случайное название файла и отправляет его к бэкенду. Файлы для проведения тестирования выбраны так, чтобы ПМ отвечал на них статусом 200 OK и не портилась результирующая статистика, т.к. проверка идёт на нагрузку, а не на корректность парсинга. Всего файлов квитанций первого типа: 373 штуки, файлов квитанций второго типа 9646 штук, файлов квитанций третьего типа: 11729 штук, этого достаточно для рандомизации запросов, все они разного имеют разный объём. Предварительно база данных ПМ

очищена и готова к записи квитанций. Целевой RPS выбран в размере 300 req/s, по 100 req/s на каждую ручку, что значительно превышает запланированный порог.

Первая итерация запуска нагрузочного тестирования на 11 минут оказалась не полностью успешной и показала результат 98,9% успешных запросов:

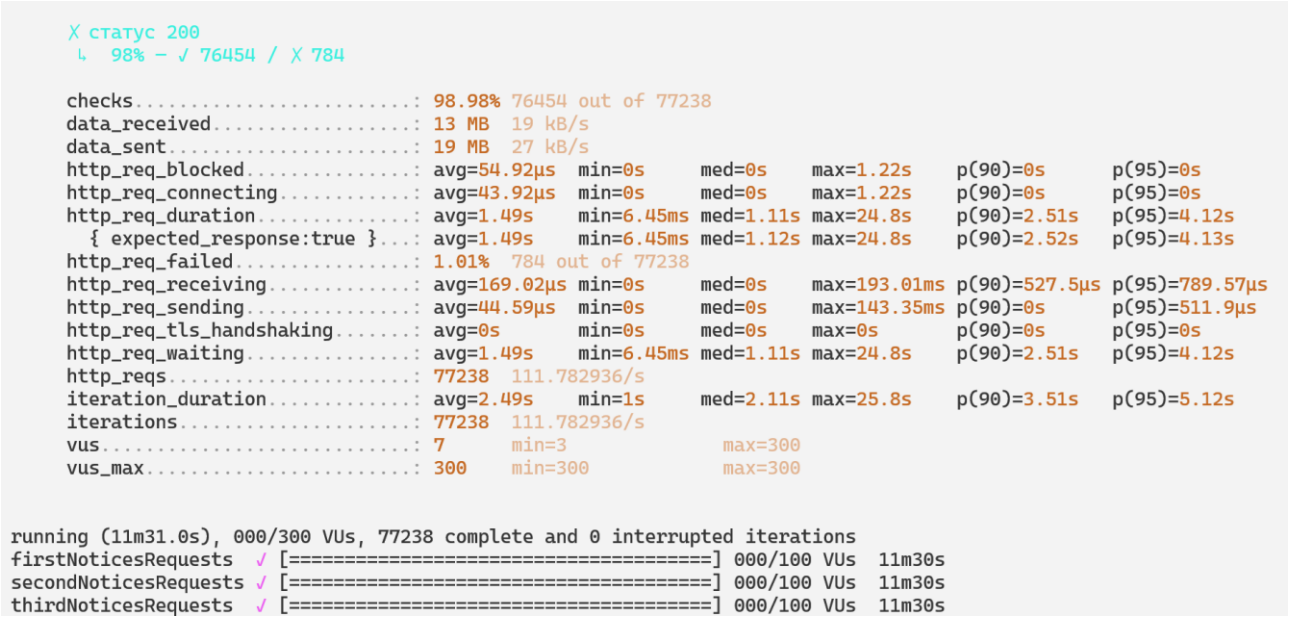

Рисунок 1 – Первый запуск нагрузки ПМ

Проанализировав ошибки в логах пода ПМ, были обнаружены некритичные проблемы:

1.Отсутствие закрытия тела реквеста r.Body.Close(). Решилась добавлением строки.

2.Отсутствие проверки на корректное считывание параметра из тела запроса. Решилась добавлением проверки на ошибку.

3.Ошибки некорректных запросов в базу. Решилась обновлением библиотеки «go-pg»[18] с версии v10.11 до версии v10.14.

После устранения проблем нагрузочный тест проведён заново, но уже на 20 минут. И по итоговому проценту его можно считать успешным:

Rates

metric	rate
checks	1/s
http_req_failed	0/s

Gauges

metric	value
vus	11
vus_max	300

Рисунок 2 – Второй запуск нагрузки ПМ

Также стоит проанализировать график RPS, «Response Time» и «Failed requests» отображённый на рисунке 3. На нём видно, что со временем на 300 req/s сервис начинает «тормозить» во времени ответа, RPS падает в среднем до 60-100 req/s, а время ответа до 10 секунд.

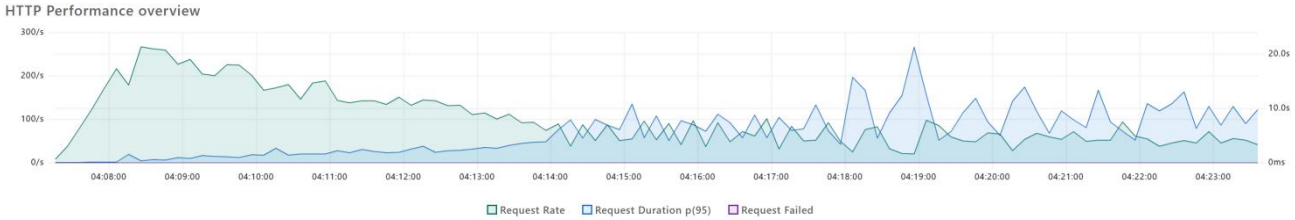

Рисунок 3 – График RPS, «Response Time» и «Failed requests» ПМ

Этот тест показывает, что если снизить нагрузку до 100 req/s, то ПМ способен на одном поде это значение RPS, имея приемлемое время ответа. Поэтому порог в заявленные 60 RPS можно считать пройденным. Однако для обеспечения меньшей нагрузки на инстанс приложения и понижения времени ответа до минимального – стоит добавить еще 2-3 пода и выделить столько же ядер процессора, при учёте доступности этих ресурсов.

3.3. Проведение нагрузочного тестирования «второго» микросервиса

«Второй» микросервис также создан преимущественно для автоматизированной работы, где взаимодействие с пользователем должно быть сведено к минимуму. Однако ВМ имеет более широкий функционал и запросы на генерацию отчётов используются не редко. Поскольку у нас есть два ключевых POST-запроса: «/makeCsv» - сбор отчёта и «/reMap» - загрузка из источников, то нужно оптимизировать их логику работы для обеспечения целостности

обрабатываемых данных. Основная идея заключается в том, чтобы эти запросы не собирали отчёты напрямую, а создавали задачи на сбор отчётов, помещая их в общую очередь. Такой подход позволит избежать долгого ожидания ответа от сервера в случае работы системы с большим количеством данных. Также для GET-запросов придумаем систему кэширования результатов завершённых задач, чтобы не нагружать каждым запросом базу данных. Изменения в логике работы представлены на рисунках 4 и 5.

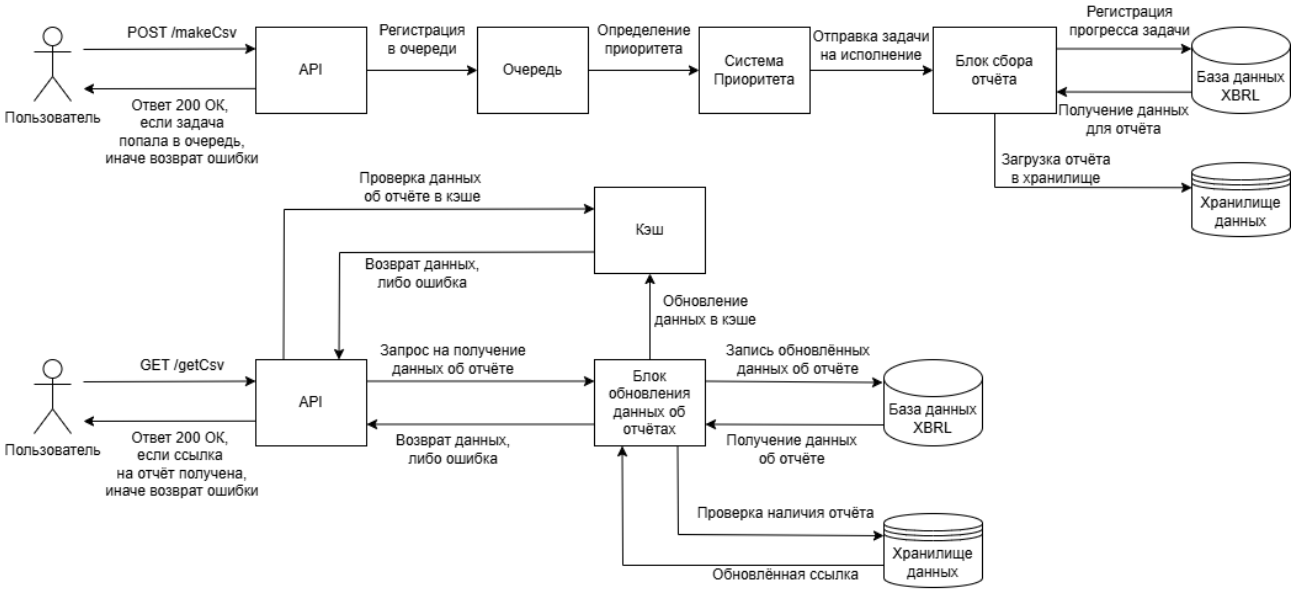

Рисунок 4 – Сбор и получение отчёта с учётом высокой нагрузки

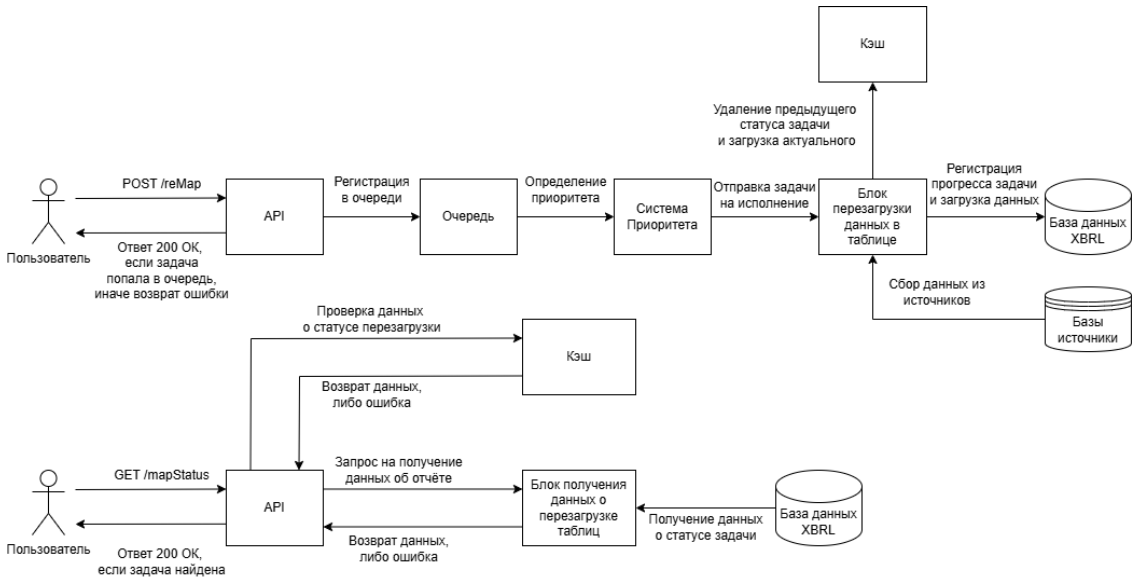

Рисунок 5 – Перезагрузка таблиц целевой БД с учётом высокой нагрузки

Данный микросервис имеет потенциал встраивания в развивающуюся платформу организации, поэтому потенциальное количество одновременных пользователей при худшем сценарии – это все сотрудники организации со стороны бизнес-заказчиков. Их примерное количество – около 300 человек, соответственно если каждый сотрудник делает запрос раз в секунду, то сервис должен выдерживать 300RPS. Поскольку GET-запросы на просмотр готовых отчётов или таблиц происходят чаще чем POST-запросы на их сбор, то примем, что POST-запросы на загрузку будут составлять 25% относительно числа GET-запросов на просмотр. Это один из худших сценариев, учитывая то, что при штатной эксплуатации приложение должно собирать отчёты раз в квартал (7 POST-запросов в 4 месяца).

Далее разберём код нагрузочного сценария, листинг которого представлен в приложении 2. В нём есть основной сценарий плавного поэтапного увеличения RPS до заданного значения для GET- и POST-запрсоов:

```
getCsv: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 80 },
                { duration: '30s', target: 400 },
                { duration: '20m', target: 400 },
                { duration: '20s', target: 0 },
            ],
            exec: 'getCsv',
        },
makeCsv: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 100 },
                { duration: '20m', target: 100 },
                { duration: '20s', target: 0 },
            ],
            exec: 'makeCsv',
        },
```

Аналогично нагрузочному тесту для ПМ в коде данного скрипта также есть функции для отправки запросов к бэкенду. Единственное допущение для данного нагрузочного теста будет заключаться в том, бэкенд не будет отдавать ошибки по

запросам «/mapStatus» и «/getCsv», если данных о сборе отчёта или таблицы ещё не было загружено в базу. Но если они появились – поддерживается стандартная работа. Это происходит потому, что эти GET-запросы идут после выполнения соответствующих POST-запросов, что портит статистику результата и не является показательным.

Как и в прошлом нагрузочном тесте, выбрано значение RPS сильно большее, чем целевое – 1000req/s, для проверки критических мощностей приложения. Как видно на рисунке 6 проблем и ошибок у сервиса не обнаружено. А вывод из рисунка 7 следует, что под нагрузкой в 1000req/s, приложение держит нагрузку стабильно, а показатель времени ответа располагается в пределах 1мс и 1сек, что является приемлемым значением.

Rates

metric	rate
checks	1/s
http_req_failed	0/s

Gauges

metric	value
vus	2
vus_max	1k

Рисунок 6 – Результат нагрузки в 1000RPS на ВМ

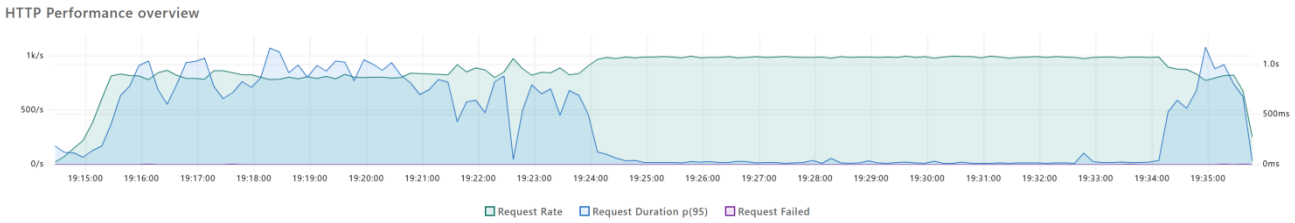

Рисунок 7 – График RPS, «Response Time», «Failed requests» для 1000 RPS на ВМ

Поскольку под нагрузкой в 1000 req/s приложение работает довольно стабильно – было принято решение увеличить количество RPS. Однако, хоть тесты на 1600 req/s и 1300 req/s завершились успешно и без ошибок, но анализ графика RPS, «Response Time» и «Failed requests» показал, что со временем вырисовывается тренд на деградацию времени ответа, что видно на рисунках 8 и 9 для 1600 req/s и 1300 req/s соответственно.

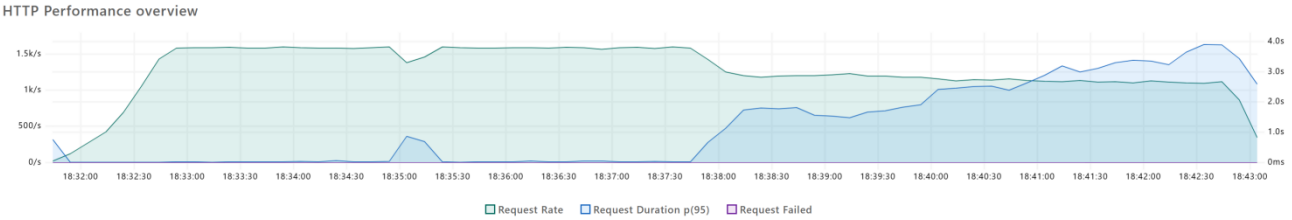

Рисунок 8 – График RPS, «Response Time», «Failed requests» для 1600 RPS на ВМ

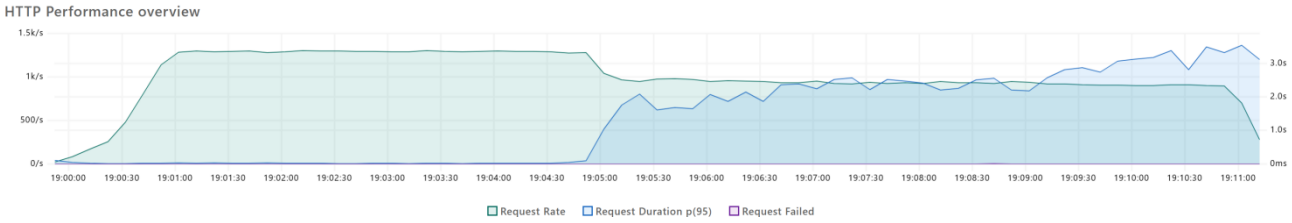

Рисунок 9 – График RPS, «Response Time», «Failed requests» для 1300 RPS на ВМ

Следовательно можно сделать вывод, что оптимальное пиковое RPS, которое приложение может стабильно держать равно примерно 1000RPS. Однако добавление одного дополнительного пода и выделения ему отдельного ядра процессора – снизит время ответа и просадки сервиса. Таким образом, можно сказать, что оба микросервиса готовы к высоконагруженности и выдерживают бóльшую нагрузку, чем требуется.

4. Создание пользовательского интерфейса

4.1. Требования и выбранные технологии

Необходимость в создании пользовательского интерфейса к разрабатываемой системе происходит из процесса первых запусков в реальных условиях. Проверка корректности и целостности загруженных данных усложняется такими проблемами как: согласование доступов в базы данных для отдельных проверяющих пользователей, лишние действия при просмотре квитанций в хранилище данных, необходимость вручную описывать каждый GET- и POST-запрос для отправки его в бэкенд и т.д. Исходя из этого появилась потребность в скором написании небольшого десктопного приложения, работающее только внутри локальной сети компании и доступ к которому будет только у пользователей, проверяющих корректность отчётов.

Как упоминалось выше, приложение должно быть простым в создании, поскольку оно не является частью основного функционала, должно выводить информацию о квитанциях до и после загрузки, отображать собранные и агрегированные записи в базе и готовые отчёты, а также иметь возможности создавать задачи на сбор отчётов и перезагрузку данных в таблицах. Доступ к источникам производится через подключение к Vault хранилищу данных.

Разберём и проанализируем основные инструменты для создания настольных приложений для компьютерных ОС:

Фреймворк / Библиотека	**Преимущества**	**Недостатки**
Electron.jsx[19]	- Лёгкая кроссплатформенность из коробки (можно перенести в браузерное решение) - Javascript поддерживает любые гибкости в построении интерфейса и визуализации данных - Простота и скорость написания скриптов - DevTools для упрощения разработки	- Большее потребление RAM/CPU чем у аналогов - Не малый размер итогового приложения засчёт движка Chromium

Фреймворк / Библиотека	Преимущества	Недостатки
	- Можно использовать любую Javascript библиотеку напрямую	
Qt (C++, PyQt)[20]	- Высокая производительность и нативный UI - Мощные инструменты собственные IDE (Qt Creator) - Большая коллекция UI-компонентов	- Большая сложность языков разработки, необходимость знать специфику фреймворка - Меньше скорость разработки - Более сложная кроссплатформенность
JavaFX[21]	- Наличие кроссплатформенности - Хорошая производительность - FXML – декларативная разметка	- Java не популярна среди UI-разработчиков - Устаревший WebView - Большой runtime, сложный деплой
.NET (WPF, WinForms)[22]	- Глубокая интеграция с Windows - Коллекция UI-компонентов - Инструменты от Microsoft (VS, XAML)	- Ограничена Windows (WPF) - Сложности деплоя под Linux - Менее гибкий UI

Таблица 2 – Анализ инструментов разработки приложения

Проанализировав таблицу, можно сказать, что наиболее подходящий инструмент – это фреймворк «Electron.js». Поскольку основные требования для приложения с пользовательским интерфейсом – это скорость разработки, кроссплатформенность и интуитивный GUI, то «Electron.js» самый подходящий под задачу фреймворк.

4.2. Реализация клиента

Для работы с фреймворком достаточно создать структуру приложения с помощью команд:

```
npm init -y
npm install electron --save-dev
```

Описать в секции скрипт «electron»:

```
"scripts": {
  "start": "electron ."
}
```

И запустить через стандартную команду: `npm start`

Клиентское приложение работает на предописанном файле «main.js», в котором указываются все параметры программы как оконного приложения, а далее оставшийся функционал работает через стандартную связку веб-инструментов: HTML, CSS и JavaScript.

Опишем и разберём страницы, которые должны быть внутри клиентского приложения:

- «Просмотр готовых отчетов» – функционал возможности получить ссылки на скачивание всех отчётов, которые собирались приложением.
- «Просмотр всех квитанций» – получение на просмотр всех доступных квитанций из Ceph S3.
- «Статус обработки квитанций» – все встреченные приложением квитанции и иные файлы из Ceph S3.
- «Просмотр таблиц» – просмотр записей в таблицах, на основании которых формируются отчёты.
- «Собрать отчёт» – функционал сбора отчёта.
- «Пересобрать таблицу» – функционал перезагрузки основных таблиц.

Стартовая страница приложения с данными пунктами меню выглядит следующим образом:

Рисунок 10 – Стартовая страница пользовательского интерфейса

На странице «Просмотр готовых отчетов», рисунок 11, представлен список ссылок на конкретные готовые отчёты, при нажатии на отчёт предлагается сохранить его на компьютер.

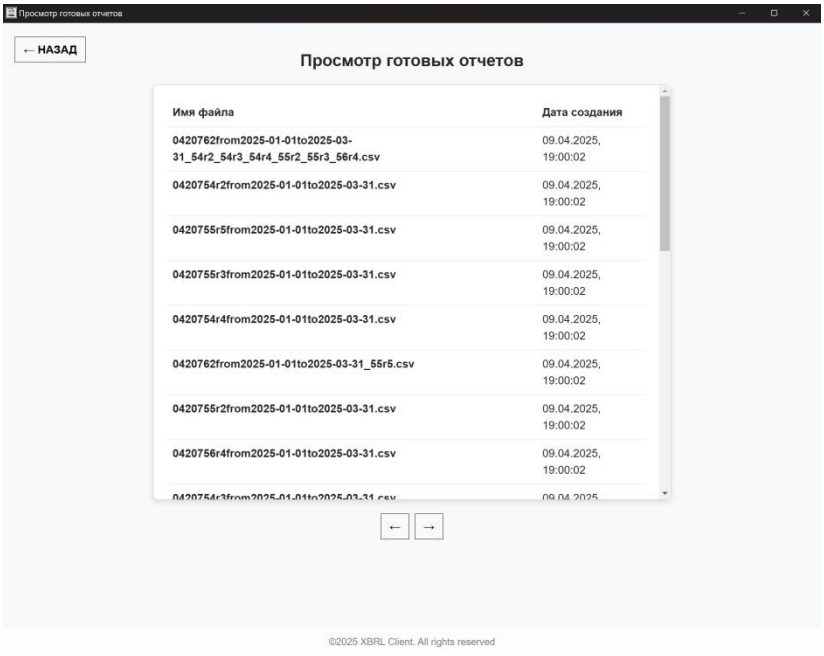

Рисунок 11 – Страница «Просмотр готовых отчетов»

На странице «Просмотр всех квитанций», отображённой на рисунке 12, представлен функционал доступа к 3-ём разным типам квитанций и возможность просмотреть их контент, как на рисунке 13.

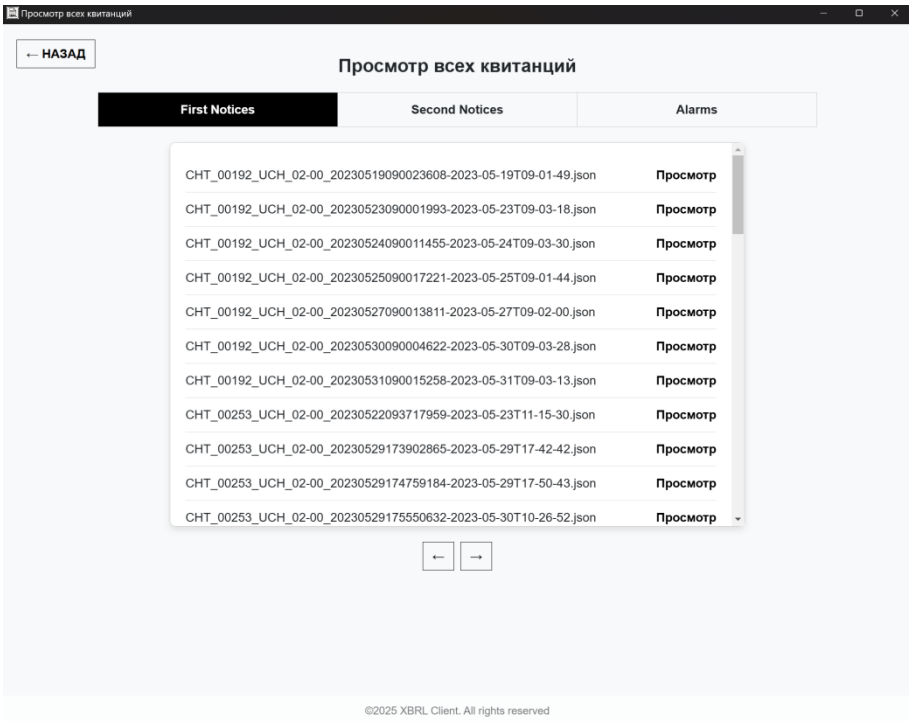

Рисунок 12 – Страница «Просмотр всех квитанций»

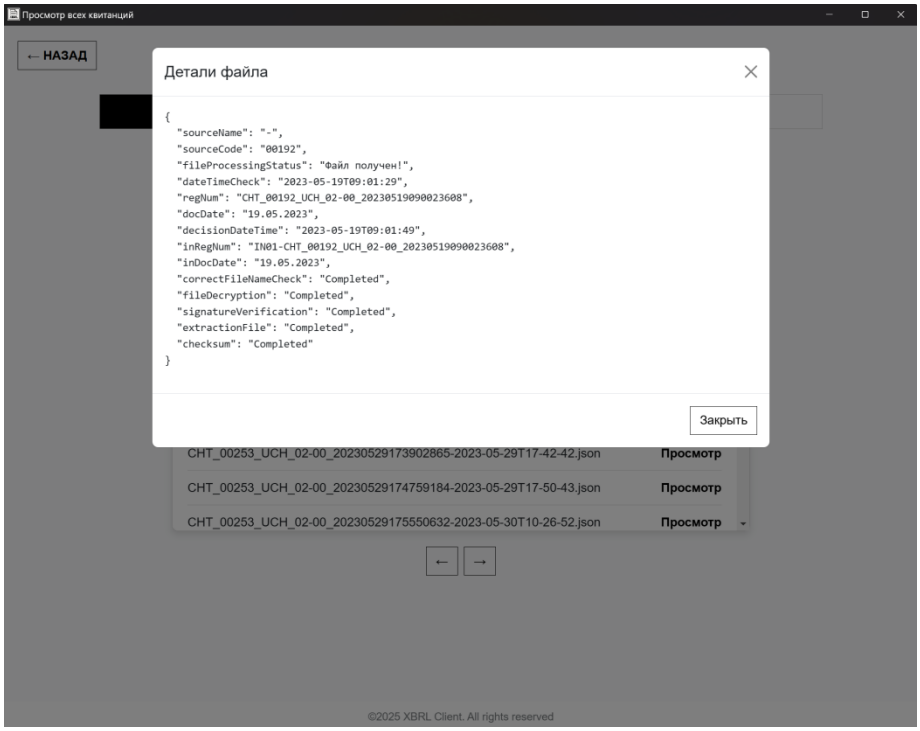

Рисунок 13 – Просмотр контента квитанции

На странице «Статус обработки квитанций», показанной на рисунке 14, располагается таблица с информацией о каждом файле: имя, статус загрузки, размер в байтах. Также на ней реализован поиск по интересующему названию файла, рисунок 15.

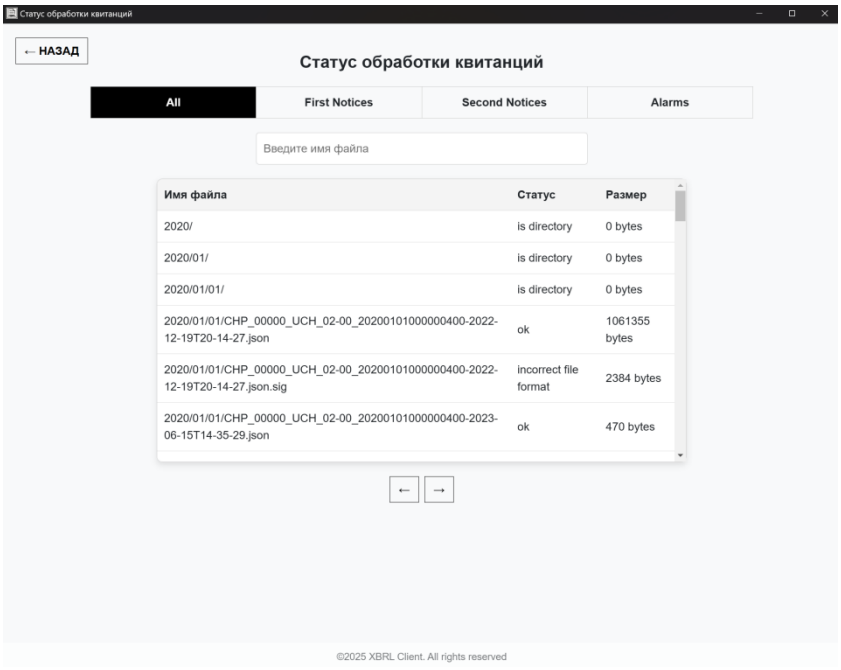

Рисунок 14 – Страница «Статус обработки квитанций»

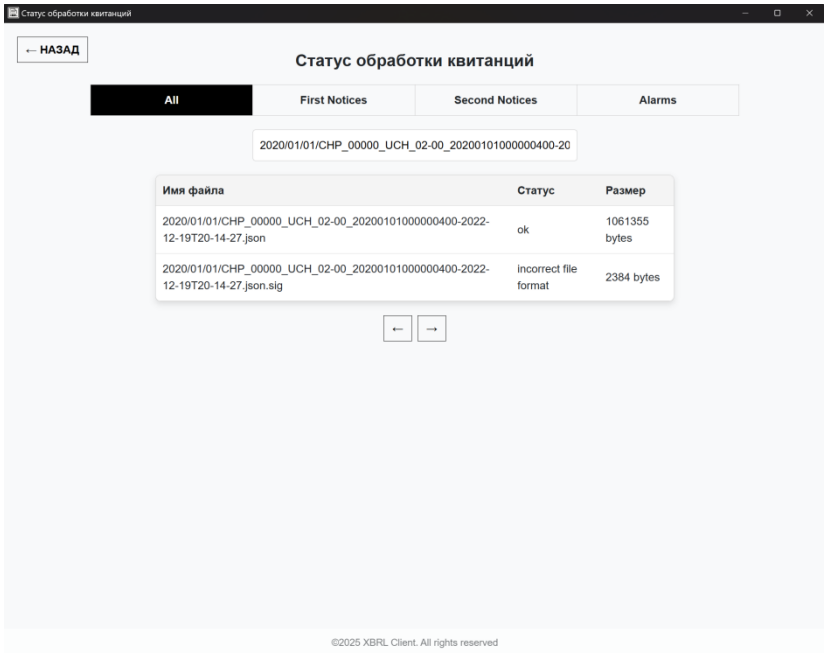

Рисунок 15 – Поиск по квитанциям

На странице «Просмотр таблиц» располагаются на выбор все таблицы, которые относятся к сборке отчёта, рисунок 16. Здесь реализован поиск по разным типам контрагентов и фильтрация по старым и новым записям.

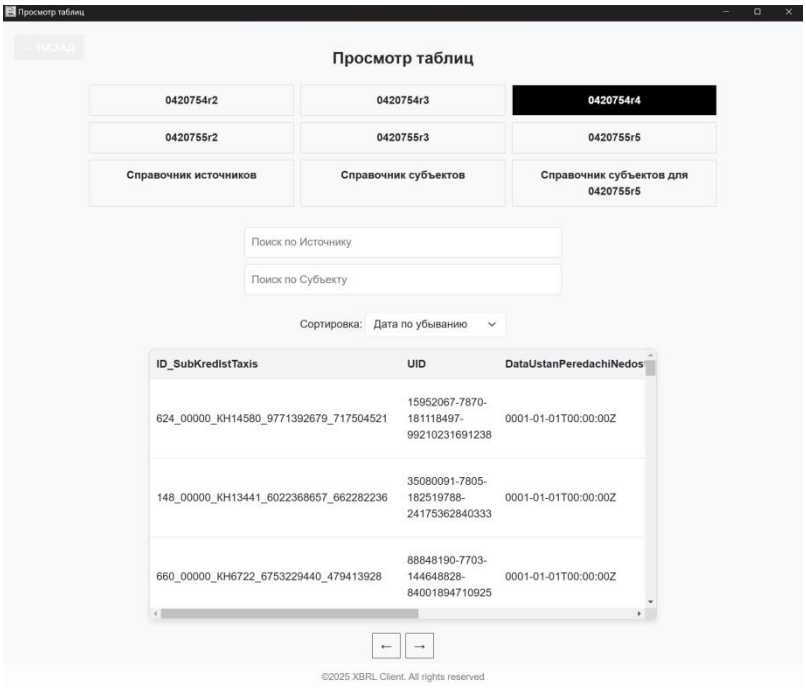

Рисунок 16 – Страница «Просмотр таблиц»

На странице «Собрать отчёт» представлен стандартный функционал сбора отчёта через вышеописанный запрос «/makeCsv», рисунок 17. На рисунке 18 представлен результат выполнения запроса: вывод ссылок на готовые отчёты (3 выбранных отчёта и справочник к ним).

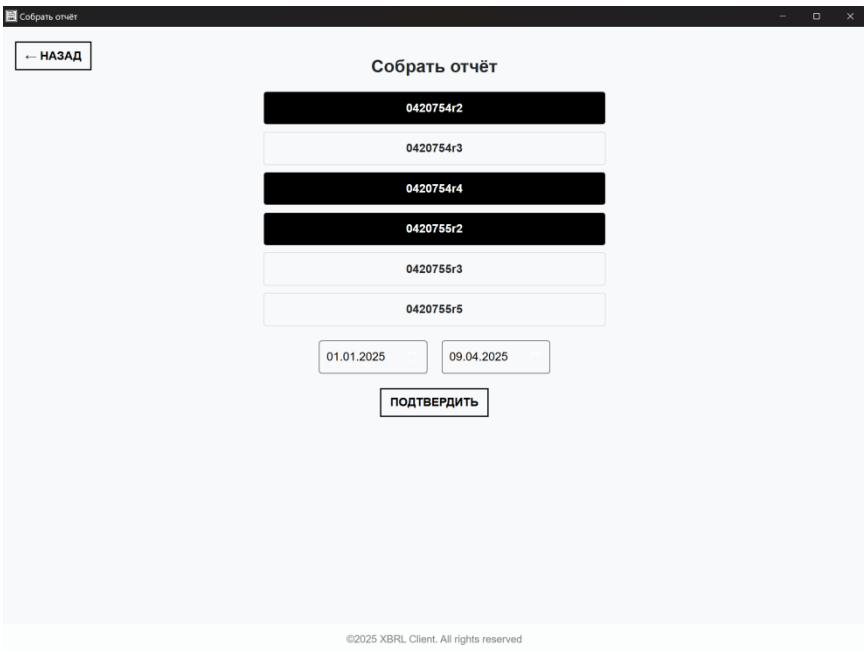

Рисунок 17 – Страница «Собрать отчёт»

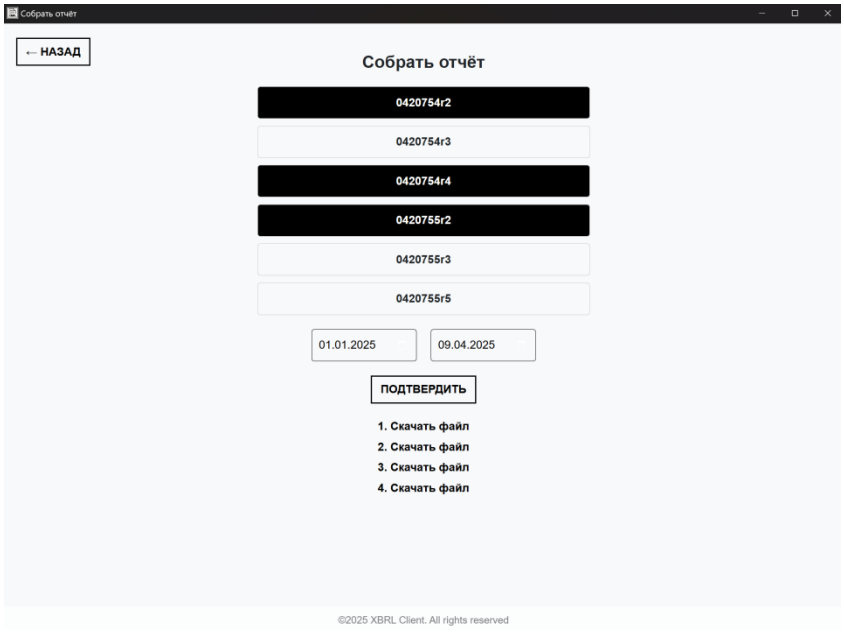

Рисунок 18 – Результат сбора отчётов.

На странице «Пересобрать таблицу» располагается возможность выбрать одну из таблиц отчётов, вывести статус загрузки данных в неё и запустить перезагрузку, рисунок 19.

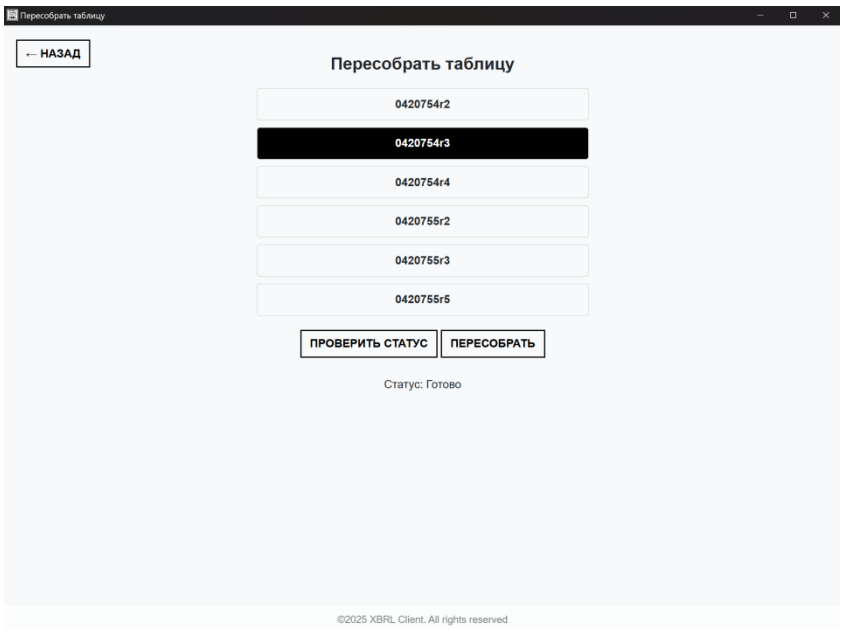

Рисунок 19 – Результат сбора отчётов.

Готовое приложение можно собрать через утилиту «electron/packager»[23], указав любую настольную ОС:

```
npx @electron/packager <sourcedir> xbrl_client --platform=<platform> --arch=<arch>
```

Таким образом, мы реализовали небольшое, но эффективное клиентское приложение с удобным пользовательским интерфейсом, что поможет проверяющим отчёты сотрудникам ускорить процесс своей работы.

Заключение

В ходе данной работы были достигнуты следующие результаты:

1. Определено целевое значение RPS каждого микросервиса. Внесены частичные доработки в сервисы для обеспечения поддержки высокой нагрузки. Подготовлено, проведено и проанализировано нагрузочное тестирование для каждого микросервиса отдельно. Это дало понимание, что сервисы стабильно выдерживают нагрузку выше требуемой: 100 req/s для ПМ и 1000 req/s для ВМ.
2. Проанализированы требования к клиентскому приложению. Выбран фреймворк «Electron.js» для разработки настольного кроссплатформенного приложения.
3. Определены необходимые пункты меню клиентского приложения, продуман его интерфейс. Разработано и протестировано пользовательское приложение.

Таким образом, данной работой удалось добиться заявленной цели.

Список источников

[1] Learn Microsoft — URL: https://learn.microsoft.com/ru-ru/dynamics365/business-central/bi-create-reports-with-xbrl (дата обращения 09.04.2025)

[2] Банк России. Открытый стандарт отчётности XBRL — URL: https://cbr.ru/projects_xbrl/ (дата обращения 09.04.2025)

[3] Банк России. Форма 0420754 «Сведения об источниках формирования кредитных историй» — URL: https://cbr.ru/explan/ot_bki/forma-0420754/ (дата обращения 09.04.2025)

[4] Банк России. Форма 0420755 «Сведения о пользователях кредитных историй» — URL: https://cbr.ru/explan/ot_bki/forma-0420755/ (дата обращения 16.02.2023)

[5] Банк России. Форма 0420762 «Реестр контрагентов» — URL: https://cbr.ru/explan/ot_bki/forma-0420762/ (дата обращения 09.04.2025)

[6] Официальная документация языка Golang. Введение. — URL https://go.dev/doc/effective_go (дата обращения 09.04.2025)

[7] Документация Ceph S3. Введение. — URL: https://docs.ceph.com/en/latest/radosgw/s3/ (дата обращения 09.04.2025)

[8] Документация PostgreSQL. Обзор. — URL: https://www.postgresql.org/ (дата обращения 09.04.2025)

[9] Документация Oracle. Базы данных. — URL: https://www.oracle.com/database/ (дата обращения 09.04.2025)

[10] Платформа Хабр. Асинхронность как основной подход к разработке высоконагруженных приложений. — URL: https://habr.com/ru/articles/733154/ (дата обращения 09.04.2025)

[11] Документация K6. Понимание результатов k6-тестирования. — URL: https://github.com/grafana/k6-learn/blob/main/Modules/II-k6-Foundations/03-Understanding-k6-results.md (дата обращения 09.04.2025)

[12] Платформа Хабр. Как построить систему, способную выдерживать нагрузку в 5 млн RPS. — URL: https://habr.com/ru/companies/ozontech/articles/749328/ (дата обращения 09.04.2025)

[13] Официальный портал Grafana. Тестирование при средней нагрузке: руководство для начинающих. — URL: https://grafana.com/blog/2024/01/30/average-load-testing/ (дата обращения 09.04.2025)

[14] Документация Grafana k6. Введение. — URL: https://grafana.com/docs/k6/latest/ (дата обращения 09.04.2025)

[15] Документация Gatling. Введение. — URL: https://docs.gatling.io/ (дата обращения 09.04.2025)

[16] Документация Locust. Введение. — URL: https://docs.locust.io/en/stable/ (дата обращения 09.04.2025)

[17] Github Документация wrk2. Обзор. — URL: https://github.com/giltene/wrk2 (дата обращения 09.04.2025)

[18] Github Документация библиотеки «go-pg». Обзор. — URL: https://github.com/go-pg/pg (дата обращения 09.04.2025)

[19] Документация Electron.js. Введение. — URL: https://electronjs.org/docs/latest (дата обращения 09.04.2025)

[20] Документация Qt. Обзор. — URL: https://doc.qt.io/ (дата обращения 09.04.2025)

[21] Документация JavaFX. Обзор. — URL: https://openjfx.io/ (дата обращения 09.04.2025)

[22] Документация Microsoft .NET WPF. Документация по Windows Presentation Foundation. — URL: https://learn.microsoft.com/ru-ru/dotnet/desktop/wpf/?view=netdesktop-9.0 (дата обращения 09.04.2025)

[23] Документация Electron/packager. О программе. — URL: https://electron.github.io/packager/main/ (дата обращения 09.04.2025)

Приложение 1

Листинг файла нагрузочного тестирования “load_notices.js” ПМ

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { firstNotices, secondNotices, thirdNotices } from './bank.js';

export let options = {
    scenarios: {
        firstNoticesRequests: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 5 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 200 },
                { duration: '18m', target: 200 },
                { duration: '20s', target: 0 },
            ],
            exec: 'firstNoticesRequests',
        },
        secondNoticesRequests: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 5 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 200 },
                { duration: '18m', target: 200 },
                { duration: '20s', target: 0 },
            ],
            exec: 'secondNoticesRequests',
        },
        thirdNoticesRequests: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 5 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 200 },
                { duration: '18m', target: 200 },
                { duration: '20s', target: 0 },
            ],
            exec: 'thirdNoticesRequests',
        },
    },
};

function getRandomElement(arr) {
    return arr[Math.floor(Math.random() * arr.length)];
}

export function firstNoticesRequests() {
    let url = ' ${apiUrlNotices}/fillDbByFirstNotice';
    let filename = getRandomElement(firstNotices)
    let payload = JSON.stringify({
        jsonFileName: filename,
    });

    let params = {
        headers: {
            'Content-Type': 'application/json',
```

```
        },
    };

    let res = http.post(url, payload, params);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}

export function secondNoticesRequests() {
    let url = '${apiUrlNotices}/fillDbBySecondNotice';
    let filename = getRandomElement(secondNotices)
    let payload = JSON.stringify({
        jsonFileName: filename,
    });

    let params = {
        headers: {
            'Content-Type': 'application/json',
        },
    };

    let res = http.post(url, payload, params);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}

export function thirdNoticesRequests() {
    let url = ' ${apiUrlNotices}/fillDbByAlarm';
    let filename = getRandomElement(thirdNotices)
    let payload = JSON.stringify({
        jsonFileName: filename,
    });

    let params = {
        headers: {
            'Content-Type': 'application/json',
        },
    };

    let res = http.post(url, payload, params);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}
```

Приложение 2

Листинг файла нагрузочного тестирования “load_xbrl.js” ВМ

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { formFiles, formNames } from './bank.js';

export let options = {
    scenarios: {
        getCsv: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 80 },
                { duration: '30s', target: 400 },
                { duration: '20m', target: 400 },
                { duration: '20s', target: 0 },
            ],
            exec: 'getCsv',
        },
        mapStatus: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 80 },
                { duration: '30s', target: 400 },
                { duration: '20m', target: 400 },
                { duration: '20s', target: 0 },
            ],
            exec: 'mapStatus',
        },
        reMap: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 100 },
                { duration: '20m', target: 100 },
                { duration: '20s', target: 0 },
            ],
            exec: 'reMap',
        },
        makeCsv: {
            executor: 'ramping-vus',
            startVUs: 1,
            stages: [
                { duration: '10s', target: 10 },
                { duration: '30s', target: 50 },
                { duration: '30s', target: 100 },
                { duration: '20m', target: 100 },
                { duration: '20s', target: 0 },
            ],
            exec: 'makeCsv',
        },
    },
};

function getRandomElement(arr) {
    return arr[Math.floor(Math.random() * arr.length)];
```

```
}

const startDate = new Date("2020-01-01T00:00:00.000Z")
const endDate = new Date("2026-01-01T00:00:00.000Z")

function getRandomDate(from, to) {
    const fromTime = from.getTime();
    const toTime = to.getTime();
    return new Date(fromTime + Math.random() * (toTime - fromTime));
}

export function getCsv() {
    let url = '${apiUrlXBRL}/getCsv?';
    let filename = getRandomElement(formFiles)
    let fromDate = getRandomDate(startDate, endDate)
    let toDate = getRandomDate(fromDate, endDate)

    let fromDateMonth = (fromDate.getMonth()+1)
    if (fromDateMonth < 10) {
        fromDateMonth = "0"+fromDateMonth
    }
    let fromDateDay = (fromDate.getDay()+1)
    if (fromDateDay < 10) {
        fromDateDay = "0"+fromDateDay
    }

    let toDateMonth = (toDate.getMonth()+1)
    if (toDateMonth < 10) {
        toDateMonth = "0"+toDateMonth
    }
    let toDateDay = (toDate.getDay()+1)
    if (toDateDay < 10) {
        toDateDay = "0"+toDateDay
    }

    let fromDateStr = fromDate.getFullYear()+"-"+fromDateMonth+"-"+fromDateDay
    let toDateStr = toDate.getFullYear()+"-"+toDateMonth+"-"+toDateDay
    url = url+"formFiles="+filename+"&fromDate="+fromDateStr+"&toDate="+toDateStr

    let res = http.get(url);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}

export function mapStatus() {
    let url = '${apiUrlXBRL}/mapStatus?';
    let filename = getRandomElement(formNames)
    url = url+"tableName="+filename
    let res = http.get(url);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}

export function reMap() {
    let url = '${apiUrlXBRL}/reMap';
    let filename = getRandomElement(formNames)
    let payload = JSON.stringify({
        tableName: filename,
    });

    let params = {
        headers: {
            'Content-Type': 'application/json',
        },
```

```
    };

    let res = http.post(url, payload, params);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}

export function makeCsv() {
    let url = '${apiUrlXBRL}/makeCsv';
    let filename = getRandomElement(formFiles)
    let fromDate = getRandomDate(startDate, endDate)
    let toDate = getRandomDate(fromDate, endDate)

    let fromDateMonth = (fromDate.getMonth()+1)
    if (fromDateMonth < 10) {
        fromDateMonth = "0"+fromDateMonth
    }
    let fromDateDay = (fromDate.getDay()+1)
    if (fromDateDay < 10) {
        fromDateDay = "0"+fromDateDay
    }

    let toDateMonth = (toDate.getMonth()+1)
    if (toDateMonth < 10) {
        toDateMonth = "0"+toDateMonth
    }
    let toDateDay = (toDate.getDay()+1)
    if (toDateDay < 10) {
        toDateDay = "0"+toDateDay
    }

    let fromDateStr = fromDate.getFullYear()+"-"+fromDateMonth+"-"+fromDateDay
    let toDateStr = toDate.getFullYear()+"-"+toDateMonth+"-"+toDateDay

    let payload = JSON.stringify({
        toDate: toDateStr,
        fromDate: fromDateStr,
        formFiles: [filename],
    });

    let params = {
        headers: {
            'Content-Type': 'application/json',
        },
    };

    let res = http.post(url, payload, params);
    check(res, {
        'статус 200': (r) => r.status === 200,
    });
    sleep(1);
}
```